ILLINOIS UNIVERSITY DEPARTMENT
OF COMPUTER SCIENCE

REPORT

ASPEN
Language Specifications

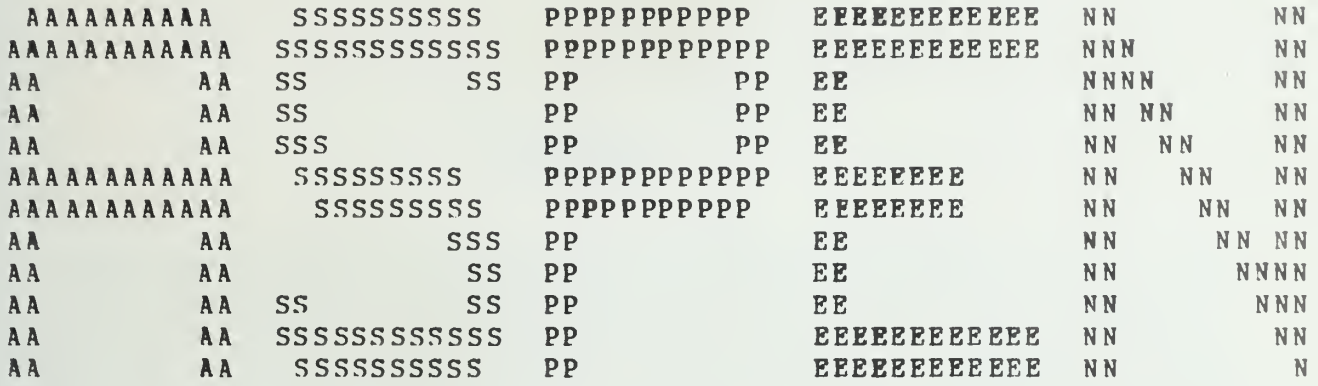
by

Thomas R. Wilcox

December, 1976

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-76-843



LANGUAGE SPECIFICATIONS

By

Thomas R. Wilcox

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

This research is supported in part by
the Research Board of the University of Illinois and
Winthrop Publishers, Inc.

December, 1976

Abstract

ASPEN is a "toy" language that has been designed for use in the teaching of compiler construction and is therefore a curious amalgam of highly refined language features that manage to barely cover the full spectrum of algorithmic languages. One selection statement is made to do the work of IF-THEN-ELSE and CASE, but with certain severe restrictions on the conditions that may be written. On the other hand, iterations must be written as a primitive infinite loop with explicit conditional exits.

Only floating point (REAL) arithmetic is supported, and although strings are present, their only practical use is restricted to the labeling of output. Both array and record aggregates are defined, but both are reduced to n-tuples of fixed-length words that are referenced through pointers; each word contains either a REAL value or a pointer to another n-tuple or string. All pointers are strongly typed.

The user is permitted to define his own types, but no coercions between types are predefined. Type definitions are implemented in the same manner as functions and procedures with all parameter passing being done by value. The uniformity of data structuring permits efficient implementation of simple parametric types and polymorphic procedures.

Using two stacks at runtime, instead of one, a degree of dynamic storage allocation is possible, but without the need for a heap and garbage collection. The "dangling reference" problem is not avoided, however. Finally, an encapsulation mechanism that allows controlled permeation of names through a program is provided to support data abstraction and information hiding, although separate compilation is not supported.

CONTENTS

ASPEN LANGUAGE SPECIFICATIONS

A.1. Textual Aspects

A.1.1. Input Format

ASPEN programs are written free form, but tokens may not cross input record boundaries. At least one blank, special character, or end of record is required to separate identifiers and numbers from each other; blanks are optional between other tokens. Comments begin with an exclamation point (!) and are terminated by the end of the input record.

The following control verbs are accepted by the compiler:

@PAGE - Start a new page in the listing.
@LMARnn - First source column is column nn (default is 01).
@RMARnn - Last source column is column nn (default is 72).
@DATA - End of program; data follows.
@RUNCHK - Enable runtime reference checking (default).
@CHEKOV - Disable runtime checking.

The right end of a control verb must be delimited in the same manner as an identifier.

A.1.2. Identifiers

A letter followed by a sequence of letters and digits constitutes an identifier. A "letter" is one of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _ # $

and the digits are

0 1 2 3 4 5 6 7 8 9

The following identifiers are reserved for special use in ASPEN and may not be used for purposes other than described in this document:

FUNC	PROC	TYPE	PACKET	IS	EXIT	END
DCL	NIL	IMPORT	EXPORT	GRANT		
IF	THEN	ORIF	ELSE	FI		
GET	PUT	LOCAL	RETURN	UNTIL	DO	OD

A.1.3. Numbers

All numeric quantities in ASPEN use a floating point representation. Literal numeric constants have the same form as unsigned PL/I DECIMAL constants.

A.1.4. Strings

Literal string constants are delimited by double quotes and if a double quote is to be part of the string, it must be written as two double quotes. String constants must fit completely within an input record, but adjacent string constants will be concatenated by the compiler to form a single string.

A.2. Declarations, Types, and Tuples

Unreserved identifiers may be used to denote constants, variables, types, procedures, parameters and special scope blocks called "packets". Each denotation of an identifier must be explicitly defined, and as in most high-level languages, each identifier has a scope of validity that is determined by the location of its declaration relative to the block structure of the program. For determining the scope identifiers, both procedures and packets are considered scope blocks. The scope of an identifier always includes the reach that contains its definition. (The reach (of a block) is the set of statements that are enclosed by a block, B, and not enclosed by any block nested within B.) The scope of the definition may be extended to

other reaches according to rules that will be explained later.

A variable is declared and initialized using the DCL statement:

DCL id: type <¬ constructor;

where "id" and "type" are identifiers and "constructor" is either an expression or a literal n-tuple constructor as defined in Sections A.2.2 and A.2.3.

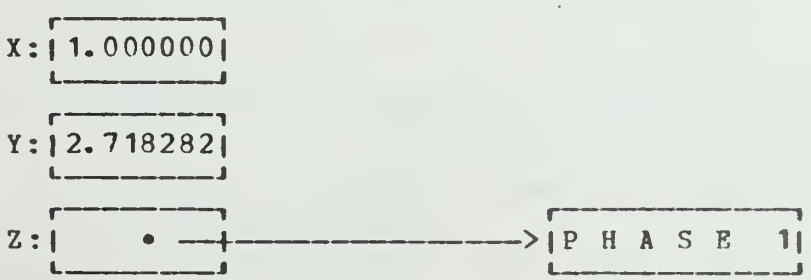
Each DCL statement allocates storage in the data area for the procedure containing the DCL and associates the identifier "id" with that storage. When the DCL statement is executed, this storage is assigned the value computed by the "constructor". The DCL statement must be the first statement executed that references the variable and may not be contained in a loop or conditional statement.

The DCL statement also declares the type of the identifier to be "type". The value of the "constructor" and all future values assigned to the variable must have this same type, and in general, ASPEN requires exact type correspondence wherever operations are performed.

ASPEN has two primitive types, REAL and STRING, so the following are legal ASPEN variable declarations:

```
DCL X: REAL <¬ 1;
DCL Y: REAL <¬ EXP(1);
DCL Z: STRING <¬ "PHASE 1";
```

The value of a string expression, and hence a string variable, is not a sequence of characters, but a pointer to the character sequence in storage. Thus the above declarations establish the following storage structure:



for The convenience of the programmer, ASPEN has many shorthand notations for its basic language constructs. Particularly, in the DCL statement, limited factoring of ":type" is permitted using the general form of the DCL statement:

DCL id1,...,idn: type <¬ constructor1,...,constructorm;

where m≤n and "constructori" is an expression rather than a literal n-tuple constructor. This is equivalent to

```
DCL id1: type <¬ constructor1;
DCL id2: type <¬ constructor2;
                 :
                 :
DCL idm: type <¬ constructorm;
                 :
                 :
DCL idn: type <¬ constructorm;
```

":type" may be omitted if the first "constructor" is a literal numeric or string constant.

A.2.1. Named Constants

Named constants (readonly variables) are also declared using the DCL statement, but using "=" instead of "<¬". For example:

```
DCL E, PI: REAL = EXP(1), 3.141593;
DCL MAXINT = 1E15;
DCL TITLE = "RESULTS OF ANALYSIS";
```

As in the declaration of variables, the constructors in the DCL are evaluated when the DCL is executed and their values become the initial value of the named constant. After the initialization, however, the named constant may not appear on the left side of an assignment or in a GET statement.

A.2.2. Records

Records (also called structures) are declared using the following form of the DCL statement:

DCL id1,...,idj bond {field1;...;fieldk};

where "fieldi" has the same form as the text permitted after DCL, that is

id1,...,idn: type bond constructors

and "bond" is either "=" for constants or "<¬" for variables.

"{field1;...;fieldk}" is called a literal record constructor, and when it is evaluated, it first creates an n-tuple in storage that has one component for each field "id" and it then initializes each component to the value of the associated "constructor". The field names within a given record must be distinct, but they may be the same as field names in other records, including any records in the initializing constructors.

Here are some sample record declarations:

```
DCL TOKEN = {NAME <¬""; INFO = {INDEX, CODE <¬ 0}};
DCL A, B, C = {RE, IM: REAL <¬ 0};
```

A field within a record is selected by following the name of the record with a dot "." and then the name of the field desired, e.g. TOKEN.NAME, TOKEN.INFO.INDEX, and TOKEN.INFO. Like PASCAL, but unlike PL/I, ASPEN requires that the field names of all ancestors of a field must be specified in the qualified reference to that field, i.e. TOKEN.INDEX and TOKEN.CODE are illegally qualified references because INFO has not been specified.

A.2.3. Arrays

Arrays are n-tuples whose components are all the same type and are referenced by an index expression rather than by a field name. The declaration for arrays is therefore similar to the record declaration and has the following form:

```
DCL id1,...,idj bond {(expression): type bond constructors};
```

"{(expression): type bond constructors}" is called a literal array constructor, and when it is evaluated it creates an array in storage. The integer portion of "expression" specifies the length (upper bound) of the array (and hence must be type REAL) and the constructors specify the initial value of the array elements. The number of initializing constructors must be no more than the length of the array, and if fewer constructors are specified, the last constructor is repeatedly evaluated until all array elements have been initialized. All constructors must compute values of the same type.

For example, a 10-component vector initially all zero is declared

```
DCL VEC = {(10): REAL <¬ 0};
```

A (constant) table of the days of the week is declared

```
DCL WEEKDAY = {(7) = "MONDAY", "TUESDAY", "WEDNESDAY",
     "THURSDAY", "FRIDAY", "SATURDAY", "SUNDAY"};
```

Arrays of arrays and arrays of records are permitted, but in that case, there can be only one initializing n-tuple constructor and ":type" must be omitted. Alternatively, for multi-dimensional arrays, a list of upper bound expressions separated by commas may be used. The constructors in the array declaration initialize the entire array in row-major order--i.e. the rightmost subscript varying the fastest. For example, the representation of a chess board could be declared

```
DCL BOARD = {(8) = {(8) <- 1, 2, 3, 4, 5, 6, 7, 8}};
```

or

```
DCL BOARD = {(8,8) <- 1, 2, 3, 4, 5, 6, 7, 8,
                      1, 2, 3, 4, 5, 6, 7, 8,
                      1, 2, 3, 4, 5, 6, 7, 8,
                      1, 2, 3, 4, 5, 6, 7, 8,
                      1, 2, 3, 4, 5, 6, 7, 8,
                      1, 2, 3, 4, 5, 6, 7, 8,
                      1, 2, 3, 4, 5, 6, 7, 8,
                      1, 2, 3, 4, 5, 6, 7, 8};
```

An element of an array is referenced by qualifying the array name with a REAL expression enclosed in braces--e.g. VEC{1} or WEEKDAY{I}. The REAL expression is truncated to an integer before it is used as the subscript. If the array name is already subscripted, the subscript expressions may be enclosed in the same set of parentheses separated by commas--e.g. BOARD{8}{J} and BOARD{8,J} are equivalent.

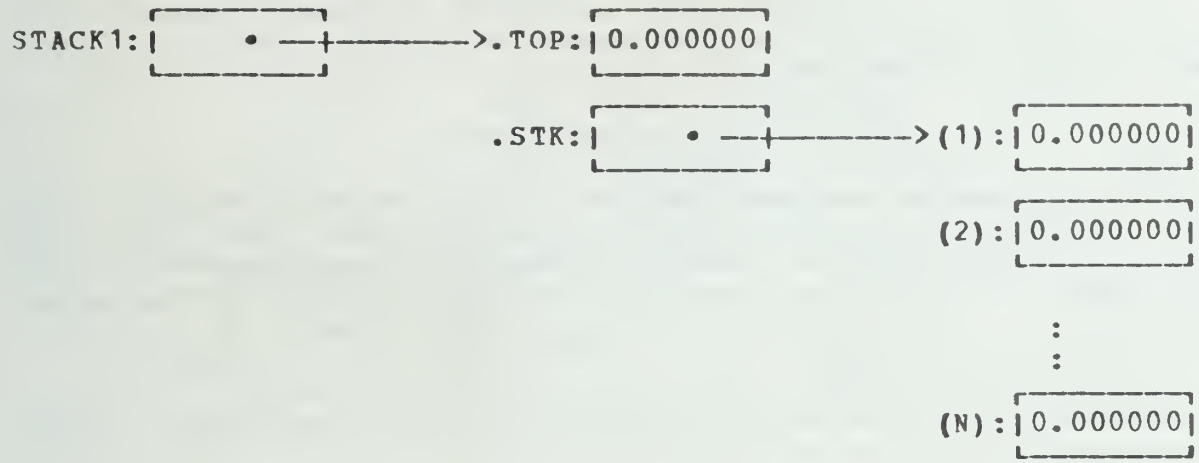
The first element of an array has index value one. The number of elements in an array may be obtained by the reference "array-name.SIZE". If the control verb @RUNCHK is in effect, code will be generated to insure that all array references are in the proper range.

A.2.4. Pointers

The value of a record or array constructor is a pointer to the storage created for the n-tuple. Hence, identifiers declared using these constructors actually denote typed pointers to storage. These n-tuple pointers are either variable or constant depending on whether <- or = was used to bind the address of the n-tuple to the identifier. For example, the storage created for STACK1, declared

```
DCL STACK1 = {TOP <- 0; STK = {(N) <- 0}};
```

would look like this:



STACK1 and STACK1.STK are constant pointers.

Pointers may be passed to procedures, returned by functions, assigned to other variables, compared for equality and used in conditional expressions, but only if the pointers involved have the same type. Note, however, that no ":type" clause is permitted when declaring a pointer using a literal n-tuple constructor. In ASPEN, each textual instance of a literal n-tuple constructor defines a unique type, which becomes the type of the n-tuple pointer.

Consider, for example, the declarations

```
DCL A, B, C <¬ {(10)<¬0};
DCL AA, BB CC <¬ {(10)<¬0};
DCL X, Y, Z <¬ {RE, IM<¬0};
DCL XX, YY, ZZ <¬ {RE, IM<¬0};
```

which define twelve pointer variables of four different types. Although A and AA are defined to have the same structure, they do not have the same type because they are initialized by different textual instances of the array constructor. Therefore, the values of A and AA cannot be compared or interchanged in an ASPEN program. On the other hand, A, B, and C have the same type, so they may be used interchangeably.

The uniqueness of literal n-tuple constructors and the exact type correspondence required by ASPEN severely limit the programs that can be written to manipulate literally defined arrays and records--all n-tuples to be manipulated must have been defined in the same DCL statement. The manipulation of array and record elements, however, is not limited by these rules. Since A{1} and AA{3} and X.RE are all of type REAL they may be combined in expressions with each other and with other values of type REAL.

A.2.5. User-Defined Types

For greater flexibility and improved program readability, literal constructors should be given meaningful names using the TYPE definition statement:

TYPE id: type = constructor;

The TYPE statement declares "id" to be a function that, each time it is invoked, creates an object that has the same structure as the objects created by the "constructor" but has type "id". The value returned by the type function is the value computed by the "constructor"--*i.e.* a REAL number or a pointer to a STRING or n-tuple.

Type functions may also be declared with parameters using the syntax

TYPE id(id*1*: type*1*;...; id*n*: type*n*): type = constructor;

where "type*i*" is either REAL, STRING, or an identifier declared in a TYPE statement. In all invocations of the type function, the type of each argument must match the "type" declared in the TYPE statement. (If several parameters are to have the same type, the type name may be factored out of the parameter list in the form "i1,id,...:type".) Parameters may be used in any context that is valid for a variable of the same type as the parameter. Some examples are

```
TYPE INTEGER(I: REAL): REAL = FLOOR(I);
TYPE REALARRAY(N: REAL) = {(N) <¬ 0};
TYPE CPLX(X, Y: REAL) = {RE, IM: REAL <¬ X, Y};
```

After a type function has been declared, its name can be used to declare the type of an identifier and the function itself can be used as a constructor in subsequent DCL or TYPE statements or as a normal function in an expression. For example,

```
DCL N1, N2: INTEGER <¬ INTEGER(N2), INTEGER(N3);
DCL A, B, C: CPLX <¬ CPLX(0,0);
DCL X: REALARRAY <¬ REALARRAY(100);
TYPE CPLXVEC(N: REAL; X: CPLX) = {(N): CPLX <¬ X};
```

Note the two roles of the type name in the above examples; to the left of <¬ the type name is declaring the type of the identifier(s) to its left, while to the right of <¬ the type name is invoking the function to create an object of that type. To avoid unnecessary writing, ":type-name bond type-name..." may be abbreviated "bond type-name...". (Note that this abbreviated form of type specification must be used with literally defined arrays and records.)

The components of an n-tuple that has been created by a type function are referenced in the same manner as the components of a literally defined n-tuple--e.g., we may write A.RE or X{1} based on the above definitions.

The only operations defined on user-defined types are assignment, identity comparison, parameter passing and conditional selection. In all operations there must be exact type match; there are no "mixed mode" expressions in ASPEN. Even types defined using REAL constructors are not considered to match type REAL. ASPEN does provide, however, a mechanism for writing conversion procedures should the programmer wish to convert between defined types (see discussion of REP below).

An important and frequent use of the ASPEN TYPE definition is to bind the unique type generated by a textual instance of a literal n-tuple constructor to an identifier that can be used many places in a program. Once a structure has been given a user-defined type name, separate instances of the structure can be declared in separate DCL statements using the name both as type declarer and structure constructor, and all instances will have the same type. For example,

```
DCL Y <¬ REALARRAY(50);
```

Declares a REALARRAY pointer that can be compared against and assigned to the REALARRAY pointer X declared above.

Type parameters are not considered in determining proper type agreement; only the type names must match. For example, a variable of type REALARRAY may be assigned a REALARRAY of any size.

A.2.6. The Special Constructor: NIL

To permit the definition of recursive data structures, ASPEN provides the constructor, NIL, which creates a null value whose type matches any ASPEN type. Using NIL, a representation of a binary tree might be defined

```
TYPE TREE(S: STRING) =
    {NAME: STRING <¬ S; LEFT, RIGHT: TREE <¬ NIL};
```

Without NIL, TREE could not be defined recursively, since there would be no way to stop the generation of TREEs once the recursive TREE function had been invoked. Using this type function, the computation tree, CT, for "A+B*C" could be constructed as follows:

```
DCL CT <¬ TREE("+");

CT.LEFT <¬ TREE("A");
CT.RIGHT <¬ TREE("*");
CT.RIGHT.LEFT <¬ TREE("B");
CT.RIGHT.RIGHT <¬ TREE("C");
```

Each reference to TREE, as a function, creates a new 3-tuple and returns a pointer to it. Since the pointer is of type "TREE" it may be assigned to fields LEFT and RIGHT of existing TREE records.

In a recursive data structure the field qualification may be arbitrarily long, but it is illegal to attempt to qualify a null pointer. The value returned by NIL is unique and any field may be tested against this value at runtime to avoid the illegal qualification of a null pointer. This test is performed automatically if the control verb @RUNCHK is in effect. Although intended to facilitate the declaration of recursive data structures, the NIL operator is not restricted to use in recursive type definitions. NIL is a value that is valid anyplace a constant is valid. It always assumes the type necessary for the context in which it is used.

":type <¬ NIL" may be shortened to ":type". For REAL types, NIL is equal to 0; for STRING types, NIL is equal to "".

A.2.7. The REP Qualifier

Because of the strict type rule, the ASPEN programmer occasionally wants to treat objects of a defined type T as if they were of the type returned by the constructor in the definition of T. For this purpose, the field name qualifier REP (for REPresentation) may be used. For example, if we have the declarations

```
TYPE STACK = LIST;
DCL X <¬ STACK;
```

and wish to pass X to procedure F written for arguments of type LIST, we must write F(X.REP) to avoid a type conflict between X and the requirements of F. REP may be used as often as desired to successively peel off the type names applied to an object. For example, if

```
TYPE LIST = REALARRAY(20);
```

then X.REP.REP is of type REALARRAY. REP may also be combined with other qualifiers. For example, if we have the declaration:

```
DCL AX = {(10) <¬ STACK};
```

then AX{1}.REP is a LIST, AX{I}.REP.REP is a REALARRAY, and AX{I}.REP.REP.SIZE is 20.

A.2.8. Functions

The general form of a function declaration is

```
FUNC id parm-list: type bond constructor IS
   declarations and statements
   END id;
```

which declares "id" to be a function that returns values of type "type". ":type" or "bond constructor" may be omitted in accordance with the rules covered in Sections A.2 and A.2.6. The returned value of the function is initially the value of "constructor", but it may be changed through references to "id.VAL" as a variable (unless the "bond" was "=").

ASPEN permits the abbreviation

```
FUNC id parm-list: type= constructor;
```

for

```
FUNC id parm-list:type = constructor IS END id;
```

For example, SQRT might be written:

```
FUNC SQRT(X: REAL): REAL = EXP(LOG(X)/2);
```

A.2.8.1. Types as Functions

Note that functions in the abbreviated form differ from TYPE declarations in Section A.2.5 only in the initial keyword. In fact, ASPEN TYPEs and FUNCs are the same in all respects except that a TYPE function gives the returned value a new type name, while in a FUNC the type of the returned value is the same as the type of the constructor. Except for the implications of returned type, the preceding and following discussions about ASPEN functions apply equally to ASPEN type functions.

Treating TYPE definitions as function definitions permits arbitrarily complex initialization code to be associated with user-defined types. It also permits type definitions with side effects:

```
DCL COLOR_CODE <¬ 0;
TYPE COLOR: REAL <¬ COLOR_CODE IS
     COLOR_CODE <¬ COLOR_CODE+1;
     END COLOR;
```

Using this definition, DCL RED,GREEN,BLUE=COLOR; generates three constants with unique COLOR codes.

Note that within a type definition that renames a record, the VAL qualifier may be used with the type name to obtain the pointer value that will be returned as the value of the type function. For example, we could write

```
                    TYPE CELL(X:REAL) =
{DATA: REAL <¬ X; NEXT, BACK: CELL <¬ CELL.VAL};
```

Then, the NEXT and BACK fields of the tuples constructed by CELL would point initially to the newly created n-tuple.

A.2.8.2. Parameter Lists and Procedure Invocation

The "parm-list" is optional in the function definition, and if present, has the form:

(field1,...,fieldn)

where "fieldi" has the same form as a field in a record declaration (Section A.2.2). If a constructor appears in a parameter declaration, it indicates that the parameter is a keyword parameter; correspondence between argument and parameter is established by binding the argument to the name of the parameter at the point of call, rather than binding them by position within the argument list. The constructor is the default value of the parameter, should no argument be supplied. If the constructor is bound to the parameter using "=", then the parameter may not be changed after entry to the procedure.

Functions are invoked by any reference to the function "id" not qualified by VAL. If the function has been declared with parameters, each reference must be followed by a parenthesized list of expressions and assignments. Any expressions must come first, and their number and type must match exactly the number and type of formal positional parameters defined in the FUNC statement. Keyword arguments, if any, follow the positional arguments in the form of assignments separated by semicolons. The list of keyword argument assignments is delimited by colons inside the parenthesized argument list.

For example, suppose IOTA is defined:

```
FUNC IOTA(SIZE:REAL; INCR, START<¬1) = REALARRAY(SIZE) IS
   DCL J<¬0;

   UNTIL J>=SIZE DO
      J <¬ J+1;
      IOTA.VAL(J) <¬ START;
      START <¬ START + INCR;
      OD;

   END IOTA;
```

Then the assignment X<¬IOTA(10) generates REALARRAY {(10)=1,2,3,4,5,6,7,8,9,10} and assigns its address to X; IOTA(5:INCR=10:) generates {(5)=1,11,21,31,41}.

Arguments are passed by value, but remember that the value of an n-tuple or string expression is a pointer to an object and not the object itself. Since arguments are passed by value, the formal function parameters are equivalent to local variables that are initialized by assignment from the corresponding argument at the time of invocation. Output from a function is restricted to the value returned by a function and the side effects of assignments to qualified parameters.

If a function returns a pointer, its invocation may be qualified with a field name or subscript list. For example, IOTA(100: START=400; INCR=-2:){4} is 394.

A.2.8.3. Procedures

The general form of a procedure declaration is similar to a FUNC declaration and has the form:

```
PROC id parm-list IS
   declarations and statements
   END id;
```

As in the FUNC statement, "parm-list" is optional.

A procedure is invoked whenever its name appears as a statement. The arguments to the procedure call are specified the same way the arguments to function calls are specified.

Aside from the way PROCs are invoked and the fact that they cannot return a value, PROCs, FUNCs, and TYPEs are essentially identical. In the following discussions, we will use the term procedure to refer to these three constructs collectively.

A.3. Expressions

The forms of expressions permitted in ASPEN are listed below; operations are listed in order of decreasing precedence:

```
numeric constant
string constant
reference
NIL
( expression )
( condition -> expression : expression )
+ expression
- expression
expression * expression
expression / expression
expression + expression
expression - expression
assignment
```

where "reference" is one of the following

```
identifier
( reference <- reference )
( condition -> reference : reference )
reference . identifier
reference {expression,...}
identifier ( expression,... )
identifier (: id1,...,idn=expression;... :)
identifier ( expression,... : id1,...,idn=expression;... :)
```

The assignment operators (see Section A.4.1) associate to the right, while all other operators associate to the left. Subscripting and pointer dereferencing are performed strictly from left to right within an expression.

An identifier may represent either a variable, a constant, or a function invocation depending on the declaration of the identifier. The qualifiers used with an identifier (".", "{}", "()", and "(::)") must be consistent with this declaration as explained in Section A.2. The standard mathematical functions ABS, FLOOR, SIN, COS, LOG, and EXP are predefined in ASPEN (but the names are not reserved). These standard functions and the operations of negation, multiplication, division, addition and subtraction are defined only for quantities of type REAL and compute values of type REAL.

A.3.1. Conditional Expressions

ASPEN provides a "conditional expression" of the form

(condition -> expression : expression)

"condition" is a Boolean expression that determines the value of the conditional expression: if it is true, the expression following "->" is evaluated and becomes the value of the conditional expression; otherwise, the expression following ":" is evaluated and becomes the value of the conditional expression. The two expressions may be any type, but they both must be the same type; this is the type of the conditional expression.

"Condition" has the general form

conjunction | ... | conjunction

where a "conjunction" is

relation & ... & relation

and a "relation" is one of

expression < expression
expression <= expression
expression = expression
expression ¬= expression
expression >= expression
expression > expression

Both expressions in a relation must be of the same type, and if the relational operator is "<", "<=", ">=", or ">" they must be of type REAL. For n-tuples and strings, pointers are compared and not the object pointed to.

The expressions within a condition are evaluated from left to right, but only when evaluation is necessary to determine the truth value of the condition. An expression in a conjunction is evaluated only if all preceding relations in the conjunction are true, and a conjunction is evaluated only if all preceding conjunctions are false.

In all but the leftmost relation, the lefthand expression and the relational operator--or the lefthand expression alone--may be omitted. If a lefthand expression is omitted, the value of the last-evaluated lefthand expression is used. If a relational operator is omitted, the textually preceding relational operator is used. These conventions permit the following syntactic shorthand:

Shorthand	Equivalent Standard Form
0 = X | Y | Z	0=X | 0=Y | 0=Z
Y <10 & >5	Y<10 & Y>5
LOG(F) <1 | >2	LOG(F)<1 | LOG(F)>2 (except for extra LOG evaluation)
Z =10 | <X & Y	Z=10 | Z<X & Z<Y
F <G & >H | I	F<G & F>H | F>I
A<X & C<Y | 10	A<X & C<Y | A<X & C<10 | A>=X & A<10 (ugh!)

To avoid possible misinterpretation of conditions like the sixth example, the programmer is advised to omit all lefthand expressions if any expressions are omitted (as in the fifth example above).

A.4. Statements

The statements discussed here are permitted in the main program and in procedure and packet definitions. Statements are normally terminated by a semicolon, but the semicolon may be omitted preceding an ELSE, ORIF, END, FI, or OD.

A.4.1. Assignment

The assignment statement is written in one of the following forms:

```
leftside  <¬ expression;
leftside +<¬ expression;
leftside -<¬ expression;
leftside *<¬ expression;
leftside /<¬ expression;
```

where "leftside" is a "reference" that denotes a variable or variable field. The types of "leftside" and "expression" must be identical, and if "+<¬", "-<¬", "*<¬", or "/<¬" are used, they must be type REAL. If "<¬" is used, the value of "expression" is assigned to the location specified by "leftside". Note, however, that the value of a string or n-tuple expression is a pointer to an object and so assignment copies pointers rather than these objects.

If "x<¬" is used, operation "x" is performed using the current value of "leftside" and the value of "expression" as operands and the result is stored back into "leftside".

A.4.2. Input/Output

The input/output statements are

```
GET leftside,...;
```

and

```
PUT expression,...;
```

"Leftside" must be a variable or variable field of type REAL, but "expression" may be type REAL or STRING. GET reads numeric constants from an implementation defined standard input file. The constants may be separated by blanks or commas. PUT converts the value of REAL expressions to a decimal character representation (implementation defined) and writes them on an implementation defined standard output file. The string "NL" in a PUT statement causes a new line to be started; "FF" (form feed) starts a new page of output. Other strings are written to the output file without conversion.

A.4.3. Selection Control Statement

ASPEN has two control statements: IF for selection and UNTIL for iteration. The simplest forms of the IF statement are

```
IF condition THEN statement;...   FI;
```

and

```
IF condition THEN statement;...
             ELSE statement;...   FI;
```

These have the standard meaning. (Note: "statement;..." denotes a list of statements of the type defined in this Section, which may be omitted if desired.) If selection is based on several disjoint conditions, the form

```
IF   condition THEN statement;...
ORIF condition THEN statement;...
                 :
                 :
               ELSE statement;...   FI;
```

may be used instead of nesting IF's in the ELSE-clause. ORIF is shorthand for ELSE IF but does not require an additional FI;. The ELSE-clause is optional.

The most general form of the ASPEN selector provides a notation for case selection based on the value of a single expression:

```
IF      condition THEN statement;...
     relop targets THEN statement;...
                        :
                        :
ORIF    condition THEN statement;...
     relop targets THEN statement;...
                        :
                        :
                      ELSE statement;...   FI;
```

There may be zero or more ORIF phrases and the ELSE clause is optional. "relop" stands for one of the six relational operators and "targets" denotes an ASPEN condition without the leading lefthand expression and relational operator. Each "relop" "targets" sequence is a continuation of the preceding "condition". The rules for omitted lefthand expressions and omitted relational operators in conditions are valid in "targets" as well.

When an IF statement is executed, the conjunctions of the continued condition are evaluated in order (from top to bottom and left to right) until a conjunction is found to be true. Then the statements in the following THEN-clause are executed. If no conjunction is found to be true, the statements in the ELSE-clause are executed.

For example, the decoding of operation codes in an interpreter might be written:

```
IF OPCODE=ADD THEN ACC <¬ ACC + VAL;
          =SUB THEN ACC <¬ ACC - VAL;
          =MUL THEN ACC <¬ ACC * VAL;
          =DIV THEN ACC <¬ ACC / VAL;
               ELSE PUT "NL","INVALID OP CODE";
   FI;
```

Or the computation of a grading histogram could be written:

```
IF ABSENT{I}=YES THEN ;
ORIF SCORE{I}<50 THEN HG{1}+<¬1;
             <60 THEN HG{2}+<¬1;
             <85 THEN HG{3}+<¬1;
             <92 THEN HG{4}+<¬1;
                 ELSE HG{5}+<¬1;
   FI;
```

A.4.4. Loop Control Statement

The ASPEN UNTIL statement is designed so that all conditions for termination of the loop are named both at the beginning of the statement and at the point of exit from the loop. The name of a termination condition is called an *exit label* and is a sequence of characters enclosed in apostrophes (the sequence of characters may not include an apostrophe). The character sequence should be mnemonic, describing the significance of the particular exit from the loop--*e.g.*, 'END OF INPUT', 'ARGUMENT FOUND', 'INVALID OP CODE', or 'TOO MANY PAGES'.

Using exit labels, the general form of the UNTIL statement is

```
UNTIL exit-label1 | ... | exit-labeln DO statement;... OD;
```

which means that the statements between DO and OD are to be executed repeatedly until an *exit clause* that is labeled with one of the "exit-labeli" is executed. An exit clause is simply a THEN- or ELSF-clause with an exit label following the "THEN" or "ELSE".

For example, a linear search for X in array A between 1 and N might be programmed:

```
I<¬1;
UNTIL 'FOUND' | 'INSERTED' | 'OVERFLOW' DO
   IF I>N THEN IF N=A.SIZE THEN 'OVERFLOW';
                           ELSE 'INSERTED';
                                A{N+<¬1}<¬X;
   ORIF A{I}=X THEN 'FOUND';
   ELSE I<¬I+1;
      FI;
   OD;
```

The UNTIL-clause preceding DO, declares the loop exit labels. The scope of this declaration is just the body of the loop, and within this scope, the exit labels may not be redeclared. In addition, each exit label must appear *exactly* once within the scope of its declaration.

One abbreviated form of the UNTIL is permitted, which is notationally and semantically isomorphic to the WHILE statements in many Algolic languages. Specifically,

```
UNTIL exit-label1 |...| exit-labeli |...| exit-labeln DO
   IF condition THEN exit-labeli FI;
   statement;
        :
   OD;
```

may be written with the "condition" in the UNTIL-clause:

```
UNTIL exit-label1 |...| condition |...| exit-labeln DO
   statement;
         :
   OD;
```

Two examples of UNTIL statements are

```
   !  FIND THE GREATEST COMMON DIVISOR OF X AND Y
   GCD<¬X;
   UNTIL GCD=Y DO
      IF GCD<Y THEN Y-<¬GCD;
               ELSE GCD-<¬Y;
         FI;
      OD;
```

and

```
   !  READ AND PROCESS AT MOST 'N' VALUES
   I<¬0;
   UNTIL 'END OF INPUT' | (I+<¬1)>N DO
      GET X;
      IF X=-1 THEN 'END OF INPUT' FI;
      PROCESS(X);
      OD;
```

A.4.5. The EXIT Statement

The statement

```
EXIT identifier;
```

terminates execution of the procedure or packet identified by "identifier". The EXIT statement must be the last statement of a conditional (THEN or ELSE) clause and must be textually nested within the entity that is to be terminated.

A.5. Object Lifetimes

While an ASPEN procedure is in execution, it may acquire storage dynamically from two separate areas. The local storage area is an extension of the procedure's activation record, and any n-tuples created in this area are destroyed when the procedure returns to its caller. The return storage area survives the invocation of the procedure, and n-tuples created there are returned to the calling procedure and may persist

indefinitely.

The disposition of n-tuples in a procedure's return area depends on the environment in which the procedure is invoked. By default, if the procedure is called from a DCL statement or formal parameter list, the n-tuples in the return area become part of the local area of the calling procedure. In all other contexts, the n-tuples in the return area become part of the caller's return area. These defaults were chosen so that ASPEN variable declarations will normally act like AUTOMATIC variables in PL/I and so that ASPEN (type) functions will normally return to the invoking procedure all of the n-tuples they have created dynamically.

For example, if we define the type E_TREE and the function GENTREE as in Figure A.1, then, under the default disposition of returned n-tuples, all E_TREE n-tuples created in the execution of GENTREE will be returned to the caller, and so, if we write

```
DCL X1 :E_TREE = GENTREE(Q);
```

the linked-list structure that represents the expression tree for the expression in Q will become part of the local storage that also contains the pointer X1.

The default disposition of returned n-tuples can be overridden through use of the reserved words LOCAL or RETURN immediately preceding the invocation of a procedure. "LOCAL FORM(...)" means all n-tuples returned by FORM are to become part of the caller's local storage; "RETURN STRUC(...)" means all n-tuples returned by STRUC are to become part of the caller's return area. The LOCAL and RETURN operators apply to the disposition of all n-tuples returned in the evaluation of arguments as well as to the n-tuples returned by the procedure or type function itself. The LOCAL and RETURN operators cannot be applied directly to a literal n-tuple constructor; the n-tuple must be returned by a procedure so that the operators can then be applied to invocations of that procedure.

As another example of storage management in ASPEN, consider the procedures in Figure A.2. ADDLINK and FORM_LIST work with lists of CELLs. ADDLINK creates a new cell for value X and links it to cell L, which is presumably in a list. Since we do not want to destroy the newly created cell when ADDLINK exits, we have used "RETURN CELL(X)" to construct the cell locally referenced as C. On exit from ADDLINK, C will be destroyed, but the cell it references will not.

FORM_LIST reads N numbers from a card and forms them into a linked list using ADDLINK. The cell created in the FUNC statement is a temporary header used to form the list. Since it is discarded before FORM_LIST exits, it should be destroyed on exit, and so it is constructed as a LOCAL CELL.

```
TYPE E_TREE(D: STRING; LT, RT: E_TREE)=
     {NAME: STRING <¬ D; LEFT, RIGHT: E_TREE <¬ LT, RT};

TYPE EXPRESSION(N: REAL) = {(N): STRING};

FUNC GENTREE(A: EXPRESSION; P<¬1): E_TREE = EXPR IS

     FUNC EXPR: E_TREE <¬ TERM  IS  ! TERM (("+" | "-") TERM)*
          DCL OP: STRING;

          UNTIL (OP<¬A{P})¬="+" & "-" DO
                P+<¬1;
                EXPR.VAL <¬ E_TREE(OP,EXPR.VAL,TERM);
                OD;
          END EXPR;

     FUNC TERM: E_TREE IS  ! "STRING" | "(" EXPR ")"
          IF A{P}="(" THEN
             P+<¬1;
             TERM.VAL <¬ EXPR;
          ELSE TERM.VAL <¬ E_TREE(A{P},NIL,NIL);
             FI;
          P+<¬1;
          END TERM;
     END GENTREE;
```

Figure A.1. An Expression tree Generator in ASPEN

No movement of n-tuples is required to implement ASPEN RETURN storage. The n-tuples can always be created in their "final resting place", which will always be in some local storage area designated by the program. Since all RETURN storage is part of some procedure's local storage it will be deleted when that procedure exits. The RETURN storage of the main program is equivalent to the heap or dynamic storage available in PASCAL, PL/I, Algol 68, and other languages, and will be deleted when the program terminates (the assumption being that the main program is invoked with a LOCAL prefix and the invoking procedure returns to the operating system when the main program returns). The design of ASPEN should reduce the need for garbage collection. Subsystems (procedures) can be written with the use of heap storage in mind, but then, through judicious use of a LOCAL prefix when the subsystem is invoked, the "heap" used by the subsystem can be deleted when the subsystem finishes processing.

A procedure has no control over the lifetime of data structures generated in its RETURN area, but then it has no way of retaining pointers to that area either. The procedure must rely on its parameters or global variables to maintain communication with n-tuples in the RETURN storage, and these links come from the invoking block, which appropriately has control over the lifetime of the storage returned by the

```
TYPE CELL(X: REAL) =
       {DATA: REAL <¬ X; NEXT, BACK <¬ CELL.VAL};

PROC ADDLINK(L: CELL; X: REAL) IS
     DCL C: CELL <¬ RETURN CELL(X);

     C.NEXT <¬  (C.BACK <¬ L).NEXT;
     L.NEXT.BACK <¬ L.NEXT <¬ C;
     END ADDLINK;

FUNC FORM_LIST(N: REAL): CELL <¬ LOCAL CELL(0) IS
     DCL X: REAL <¬ 0;
     DCL P: CELL <¬ FORM_LIST.VAL;

     UNTIL (N<¬N-1)<0 DO
           GET X;
           ADDLINK(P,X);
           P <¬  P.NEXT;
           OD;

     FORM_LIST.VAL <¬ FORM_LIST.VAL.NEXT;
     FORM_LIST.VAL.BACK <¬ P;
     END FORM_LIST;
```

Figure A.2. Sample Use of Storage Control

procedure.

A.6. Packets

A packet is the ASPEN encapsulation mechanism that allows the programmer explicit control over the scope of names in his program. Packets are permitted wherever declarations are permitted and have the following structure:

```
PACKET id IS
   IMPORT and EXPORT statements
   statements and declarations
   END id;
```

The normal Algol 60 scope rules apply to ASPEN procedures--i.e., the scope of an identifier is automatically extended to the reach of a procedure lying in the scope of the identifier unless the identifier is redefined within the reach of that

procedure. In the case of PACKETs, however, names are not automatically inherited from the surrounding environment. The names defined outside a packet that are to be used inside the packet must be explicitly imported into the packet using an IMPORT statement of the form:

IMPORT name1,...,namen;

The IMPORT statement extends the scope of "namei" to include the reach of the packet containing the IMPORT statement.

Names defined inside the packet can be referenced outside the packet, but only if they are explicitly listed in an EXPORT statement within the packet. An EXPORT statement has the same form as an IMPORT statement:

EXPORT name1,...,namen;

The EXPORT statement extends the scope of "namei" to include the reach of the immediately enclosing packet or procedure.

If a name is to be exported (imported) several levels from its definition, it must be referenced in an EXPORT (IMPORT) statement at each level of packet nesting. Names cannot be exported across procedure borders. And above all, for each identifier in the program, there must be exactly one definition of the identifier in each disjoint scope of the identifier.

A packet may import an name only if it is granted permission to do so by a GRANT statement of the form:

GRANT id name1,...,namen;

where "id" is the packet "id", and the "namei" are the only names that may be IMPORTed by the packet. These names must be defined in the block containing the GRANT--either through a declaration there or because they were IMPORTed or EXPORTed into the block. There must be exactly one GRANT statement for each packet appearing in a program, it must be located in the procedure or packet that contains the packet, and it must textually precede all statements that reference the names exported from the packet.

When the GRANT statement is executed, the packet is initialized--the declarations and statements in the packet are executed in sequence to construct storage and initialize the variables and constants in the packet. Packet and procedure definitions that reference identifiers exported from a packet may precede the GRANT statement for the packet, but these blocks may not be executed before the packet has been initialized.

The storage associated with a packet is an extension of the storage for the procedure containing the packet. The lifetime of variables and constants declared in the packet is the lifetime of the surrounding procedure; n-tuples are either

retained in or returned from that procedure.

In order for two packets to share exclusive use of a resource (name), three elements must be present: first, the packet providing the resource must EXPORT the name of the resource; second, the packet(s) using the resource must IMPORT the name; and third, the block enclosing both packets must GRANT the exported name to the packet(s) wishing to import it. The enclosing block should be viewed as a manager of the two packets that wish to communicate, and then the GRANT statement becomes a precise statement of the managerial decision to permit the communication requested by the IMPORT and EXPORT statements.

A.6.1. Import, Export, and Grant Specifications

Each "name*i*" specification in an IMPORT, EXPORT, or GRANT statement has one of the following forms:

```
modifier identifier
modifier identifier . name
modifier {name1,...,namem}
```

where "modifier" is optional, and if present, is a list of letters enclosed in parentheses. The first "identifier" in each name specification is the name of a variable, constant, function, procedure, or type that is to be transported across a packet border. If the identifier has been defined using a literal n-tuple constructor, then field names of the n-tuple may be transported by including them as qualifiers in the specification. Several fields of the n-tuple can be transported by enclosing the field names in braces following the qualifying dot. All field names to be transported must be fully qualified (but see the discussion of the X modifier below). The identifier "#" may be used to transport the subscripting of an array. If keyword parameters are to be transported, the function, procedure, or type name must be qualified by the (list of) keyword(s) that is to be transported. (In the case of type names, which denote both a function and possibly a literal n-tuple, both keyword parameters and subfield names are listed in the same list following the type name.)

Some examples based on previous definitions are:

```
IMPORT CELL.{DATA,(R){NEXT,BACK}}, ADDLINK, FORM_LIST;
IMPORT CPLXVEC.#, AX; !  NO SUBSCRIPTING OF AX
IMPORT TREE.{NAME};     !  IMPORT NAME FIELD ONLY
IMPORT SQRT, IOTA.{START,INCR}, COLOR;
EXPORT STACK1.(X)STK.#;
EXPORT TOKEN.(R){NAME,(X)INFO.CODE};
```

Each modifier indicates which of the possible uses of the name are to be imported, exported, or granted, but it cannot weaken the access restrictions that have already been assigned to the name by virtue of its declaration in, or transportation into, the block. If the modifier precedes a bracketed list, the modifier applies to all unqualified identifiers in the list. In the last EXPORT statement, for example, (R) applies to NAME and CODE, but not INFO.

The following modifiers are currently recognized:

Modifier	Use of Name Z Imported, Exported or Granted
I	Invocations of procedure Z are allowed.
D	Declarations using type Z are allowed.
T	Type. (Implies I and D.)
Q	Qualification of pointer (of type) Z.
R	Reading of value of (type) Z. (Also implies Q.)
U	Updating of variable (of type) Z.
V	Variable (of type). (Implies R and U.).
X	None. (Name present for qualification only.)

The modifiers applicable to each type of identifier and the default applied if none is specified is as follows:

Type of Identifier	Modifiers Allowed	Default
Scalar Variable	V, U, R	V
Scalar Constant	R	R
Keyword Parameter	U	U
N-tuple Identifiers	Q,V,U,R,X	Q
Type Names	T,I,D,X,Q,V,U,R	T,Q
Other Procedures	I	I

The X modifier indicates that the identifier that follows it may not be used in the block to which it is transported; the identifier is present in the specification only to qualify the names of some of its subfields, which are to be transported. For example, as a consequence of the EXPORT statements given above, outside the packet, STACK1{I} and TKN.ADDR must be used as "abbreviations" for STACK1.STK{I} and TKN.INFO.ADDR respectively (assuming TKN is of type TOKEN). At their destination, of course, the transported subfield names must be uniquely referencable using only the transported qualifiers.

When a type name is transported, two modifiers are applicable: one for the type name itself (I, D, T, X) and one for identifiers of that type (Q, V, U, R). For example,

```
EXPORT (T,R)TREE.{NAME}, (X,Q)CELL.{DATA};
```

exports type TREE such that only constants of type TREE may be declared or created; no identifiers of type CELL may be declared nor may cells be created outside the packet, and any identifiers

of that type that are exported, may only be used to qualify references to the DATA fields of the objects created in the packet.

The predefined qualifiers REP and SIZE must be explicitly transported for each variable, constant, or type.

A.6.2. Data Abstraction

To protect a data structure from "unauthorized" access, the data structure and a collection of certified procedures for accessing the data structure would be defined as a packet, but only the procedure names would be exported from the packet. Hence the only way to access the data structure would be to invoke one of the procedures. The implementation of a queue facility is programmed in this way in Figure A.3.

Type QUEUE, procedure ENQ (enqueue) and function DEQ (dequeue) are the only components directly accessible from outside the packet. Storage of type CELL can not be created directly in statements outside the packet, nor can these statements access the fields DATA, NEXT, HEAD, or TAIL in CELL's and QUEUE's. Since all access to the list structures for implementing queues is restricted to just the coding in packet QUEUES, it can easily be proved that the representation is correct and secure.

The ASPEN packets in Figure A.4, illustrate the degree of control the programmer has over access to his data structures.

The SYMBOL_TABLE and NAME_TABLE packets encapsulate the symbol table and name table managers of a compiler. The design of the compiler requires that the symbol table manager must have access to the STP field of each name table entry so that it can be changed to reflect the varying name-symbol associations in effect at each point in the source program. For security, the STP field should not be accessible to any other packet except the name table manager.

The ATTR field of each SYMBOL n-tuple contains language dependent attributes for the symbol. Since these do not influence the symbol table management algorithms, the field is only indirectly specified by referencing the type ATTRIBUTES in the definition of SYMBOL. The environment where the SYMBOL_TABLE packet is used must supply the appropriate definition for this type and will have unrestricted access to it.

```
PACKET QUEUES IS
   EXPORT QUEUE, ENQ, DEQ;
   TYPE CELL(X: REAL) = {DATA: REAL <¬ X; NEXT: CELL};
   TYPE QUEUE = {HEAD, TAIL: CELL};

   DCL FREE_LIST: CELL;

   PROC ENQ(Q: QUEUE; DATA: REAL) IS
        DCL C: CELL <¬ FREE_LIST;

        IF FREE_LIST=NIL THEN C <¬ CELL(DATA);
        ELSE FREE_LIST <¬ C.NEXT;
             C.DATA <¬ DATA;
             C.NEXT <¬ NIL;    FI;
        (Q.TAIL=NIL -> Q.HEAD : Q.TAIL.NEXT) <¬ C;
        Q.TAIL <¬ C;
        END ENQ;

   FUNC DEQ(Q: QUEUE): REAL IS
        DCL P: CELL <¬ Q.HEAD;

        IF P¬=NIL THEN
           DEQ.VAL <¬ P.DATA;
           Q.HEAD  <¬ P.NEXT;
           IF Q.HEAD=NIL THEN Q.TAIL <¬ NIL FI;
           P.NEXT <¬ FREE_LIST;
           FREE_LIST <¬ P;          FI;
        END DEQ;
    END QUEUES;
```

Figure A.3. A Queue Facility in ASPEN

A.7. Parametric Types

The combination of TYPEs and PACKETs in ASPEN provide the programmer with a powerful data abstraction mechanism. The data abstraction mechanism is further strengthened by the ability to define TYPEs in which component types are parameters of the definition and the ability to define procedures that can accept and manipulate the parameterized types. Certain restrictions must be placed on this feature, so that the programmer is never surprised by the amount of code that is generated by an ASPEN type or procedure definition.

```
IMPORT ERROR, ATTRIBUTES;
EXPORT (Q)SYMBOL.ATTR;
GRANT NAME_TABLE (D)SYMBOL;
GRANT SYMBOL_TABLE (D)NAME.STP, (D)ATTRIBUTES, ERROR,
                   UNIQUE_NAME;

PACKET NAME_TABLE IS
   IMPORT (D)SYMBOL;
   EXPORT (D)NAME.STP, UNIQUE_NAME;
   TYPE NAME = {NAME: STRING;  TYPE: REAL;  STP : SYMBOL};
            :
   END NAME_TABLE;

PACKET SYMBOL_TABLE IS
   IMPORT (D)NAME.STP, (D)ATTRIBUTES, ERROR, UNIQUE_NAME;
   EXPORT (D)SYMBOL.ATTR, ENTER, OPENSCOPE, CLOSESCOPE;

   TYPE SYMBOL(PTR: NAME) =
        {NTP      :NAME          <¬ PTR;
         BLOCK    :BLOCK_RECORD   = BLOCK;
         NAMESAKE :SYMBOL        <¬ PTR.STP;
         NEIGHBOR :SYMBOL        <¬ BLOCK.RESIDENTS;
         ATTR     :ATTRIBUTES    <¬ NIL};
   TYPE BLOCK_RECORD =
        {HEIR     :BLOCK_RECORD <¬ NIL;
         SIBLING  :BLOCK_RECORD <¬
                          (BLOCK=NIL->NIL:BLOCK.HEIR);
         PARENT   :BLOCK_RECORD <¬ BLOCK;
         RESIDENTS:SYMBOL        <¬ NIL};

   DCL BLOCK <¬ BLOCK_RECORD; ! CURRENT BLOCK

   FUNC ENTER(PTR: NAME): SYMBOL IS
        IF PTR.STP¬=NIL & PTR.STP.BLOCK=BLOCK
        THEN ERROR(:MSG="MULTIPLE DECLARATION":);
             PTR <¬ UNIQUE_NAME;                  FI;
        ENTER.VAL <¬ SYMBOL(PTR);
        BLOCK.RESIDENTS <¬ PTR.STP <¬ ENTER.VAL;
        END ENTER;

   PROC OPENSCOPE IS
        BLOCK.HEIR <¬ BLOCK <¬ BLOCK_RECORD;
        END OPENSCOPE;

   PROC CLOSESCOPE IS
        DCL P: SYMBOL <¬ BLOCK.RESIDENTS;
        UNTIL P=NIL DO   ! POP NAMESAKE STACKS
              P.NTP.STP <¬ P.NAMESAKE;
              P <¬ P.NEIGHBOR;  OD;
        BLOCK <¬ BLOCK.PARENT;
        END CLOSESCOPE;
   END SYMBOL_TABLE;
```

Figure A.4. A Symbol Table Manager in ASPEN

A.7.1. Superstructures

A superstructure is the framework for a class of data structures. The superstructure defines the characteristics that distinguish its members from other data structures and also defines where members of the class may have different characteristics. An array is an example of a commonly used superstructure; the essential characteristics of an array are that all elements have the same structure and each element can be referenced by an index value from a finite set; the structure of the elements of the array are not essential to the concept of an array.

To define a superstructure in ASPEN, component types of the superstructure are made parameters of the type definition. For example, the primitive array constructor can be recast as a user-defined superstructure using the following definition:

```
TYPE(TYP) ARRAY(SIZE: REAL) = {(SIZE):TYP};
```

Enclosing the identifier TYP in parentheses, preceding the superstructure name, declares it to be a type-parameter of the superstructure definition; it is implicitly of type TYPE and may be used in any ":type" construct and as a type-argument in a superstructure reference. It may not be used as a function or as a superstructure. Unlike the other parameters of the type definition, type-parameters are bound at compile time, when the parameterized superstructure name is used in a declaration or expression.

Examples of ARRAY declarations are

```
DCL AR:(REAL)ARRAY = ARRAY(100);
DCL AS:(STRING)ARRAY = ARRAY(10);
DCL AP:(STACK)ARRAY;
DCL AA:((REAL)ARRAY)ARRAY = ARRAY(5);
TYPE(X) MATRIX(M,N: REAL):((X)ARRAY)ARRAY = ARRAY(M) IS
   DCL J<¬0;
   UNTIL (J+<¬1)>M DO MATRIX.VAL(J) <¬ ARRAY(N) OD;
   END MATRIX;
```

Note that type-arguments precede the name of the superstructure, following the : in the declarations, and the type-arguments are not used when the type function is invoked.

As with objects of all types, specific instances of superstructures may be passed to procedures defined to accept them:

```
PROC SORT(A:(REAL)ARRAY) IS
   DCL I,J<¬0;
   UNTIL (I+<¬1)>=A.SIZE DO
      J<¬A.SIZE;
      UNTIL (J-<¬1)<I DO
         IF A{J}>A{J+1} THEN
            T <¬ A{J};
            A{J} <¬ A{J+1};
            A{J+1} <¬ T;
            FI;
         OD;
      OD;
   END SORT;
```

A.7.2. Polymorphic Procedures

Procedures that operate on all instances of a superstructure are called polymorphic procedures since they operate on many different forms of data structures. In ASPEN, polymorphic procedures are defined using the notation of superstructure definitions, but applied to PROC or FUNC identifiers. For example SEARCH is a polymorphic procedure that returns the index of the array element containing the value of its second argument:

```
FUNC(T) SEARCH(A:(T)ARRAY; X:T):REAL IS

   UNTIL (SEARCH.VAL+<¬1)>A.SIZE DO
      IF A{SEARCH.VAL}=X THEN EXIT SEARCH FI;
      OD;

   SEARCH.VAL<¬NIL;
   END SEARCH;
```

As with superstructures, when polymorphic procedures are invoked, type-arguments are never specified; e.g., SEARCH is called via the statement "SEARCH(AS,"TRW");".

A.7.3. Items of Parametric Type

In a polymorphic procedure (and in superstructure definitions) the operations that can be performed on items having a parametric type are restricted to the operations that can be performed on all types. In SEARCH, for example, the only operation performed on X (of parametric type T) is identity

comparison. A polymorphic array sort procedure is not definable in ASPEN since not all ASPEN types have an ordering defined for them. Polymorphic sort procedures can be defined only for superstructures having a field known to always be REAL. For example:

```
TYPE(L,R) PAIR(LV:L; RV:R) = {LEFT:L<¬LV; RIGHT:R<¬RV};

PROC(ANY) PSORT(A:((ANY,REAL)PAIR)ARRAY) IS
   DCL I,J<¬0; DCL T:(ANY,REAL)PAIR;

   UNTIL (I+<¬1)>=A.SIZE DO
      J<¬A.SIZE;
      UNTIL (J-<¬1)<I DO
         IF A{J}.RIGHT>A{J+1}.RIGHT THEN
            T <¬ A{J};
            A{J} <¬ A{J+1};
            A{J+1} <¬ T;
            FI;
         OD;
      OD;
   END PSORT;
```

A more complex example of the use of superstructures is shown in Figure A.5.

```
PACKET ASSOC_MEM IS

   IMPORT PAIR, PSORT;
   EXPORT (D)MEMORY, FETCH, STORE, MEMSORT;

   TYPE(ADR,VAL) MEMORY = (LNK:(ADR,VAL)MEMORY; AD:ADR; MV:VAL);

   FUNC(A,M) FETCH(MEM:(A,M)MEMORY; X:A):M IS
        DCL P:(A,M)MEMORY <¬ MEM;

        UNTIL 'X FOUND' | P=NIL DO
              IF P.AD=X THEN 'X FOUND';
                             FETCH.VAL <¬ P.MV;
                 FI;
              P <¬ P.LNK;
              OD;
        END FETCH;

   FUNC(A,M) STORE(MEM:(A,M)MEMORY; AD:A; X:M):(A,M)MEMORY IS
        DCL P:(A,M)MEMORY <¬ STORE.VAL <¬ MEM;

        UNTIL 'AD FOUND' | 'END OF LIST' DO
              IF P=NIL THEN 'END OF LIST'
                            P <¬ MEMORY;       ! GET NEW CELL
                            P.LNK <¬ MEM.LNK; ! ADD TO FRONT
                            MEM.LNK <¬ P;      ! OF LIST.
                            STORE.VAL <¬ P;
              ORIF P.AD=AD THEN 'AD FOUND';
              ELSE P <¬ P.LNK;
                 FI;
              OD;
        P.MV <¬ X;
        END STORE;

   FUNC(A) MEMSORT(MEM:(A,REAL)MEMORY):((A,REAL)PAIR)ARRAY IS
        DCL P:(A,REAL)MEMORY <¬ MEM;
        DCL N<¬0;

        UNTIL P=NIL DO N+<¬1;
                       P <¬ P.LNK;
                       OD;
        MEMSORT.VAL <¬ ARRAY(N);
        DCL J<¬0;    P <¬ MEM;
        UNTIL (J+<¬1)>N DO MEMSORT.VAL[J] <¬ PAIR(P.AD,P.MV);
                           P<¬P.LNK;
                           OD;
        PSORT(MEMSORT.VAL);
        END MEMSORT;
   END ASSOC_MEM;
```

Figure A.5. An Abstraction of Associative Memory

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-76-843	2.	3. Recipient's Accession No.

4. Title and Subtitle

ASPEN Language Specifications

5. Report Date
December 1976

6.

7. Author(s)
Thomas R. Wilcox

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois
Urbana, Illinois

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address

Department of Computer Science
University of Illinois
Urbana, Illinois

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

ASPEN is a "toy" language that can be used as a sample source language in the teaching of compiler construction. As such, its design incorporates language constructs that can be handled by fundamental compiler construction techniques and yet are expressive, well-structured and reasonably secure, in keeping with current trends in language design. As a result, ASPEN is goto-less, strongly typed, and provides efficient, orthogonal mechanisms for information hiding and parameterized user-defined types. ASPEN's dynamic storage allocation mechanism and its polymorphic structures and procedures can be implemented without a heap and with no run-time support routines other than those needed for format-free input and output of strings and numbers.

These language specifications define the language and offer some examples of its use.

17. Key Words and Document Analysis. 17a. Descriptors

programming languages, information hiding, data abstraction, structured control statements, efficient run-time storage management, recursive and polymorphic types and procedures

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
	20. Security Class (This Page UNCLASSIFIED	22. Price